Assignment 7 – Huffman Coding Report

Arushi Tyagi

CSE 13S – Winter 24

Purpose

Audience for this section: Pretend that you are working in industry, and write this paragraph for your boss. You are answering the basic question, "What does this thing do?". This section can be short. A single paragraph is okay.
The purpose of this assignment is to efficiently do data compression using huffman encoding and decoding. The program takes in an input text file and encodes it, then puts the encoded information into an output file. When decoding, it takes the encoded file as the input and then outputs it into the decoded text.

Questions

Please answer the following questions before you start coding. They will help guide you through the assignment. To make the grader's life easier, please do not remove the questions, and simply put your answers below the text of each question.

- Describe the goal of compression. (As a hint, why is it easy to compress the string "aaaaaaaaaaaaaaaa")
  The goal is to get a smaller version of a file without losing any data. It's easy to compress the string "aaaaaaaaaaaaaaaa" because it has low entropy which means that there are less unique characters in it. Therefore, it takes up less storage.
- What is the difference between lossy and lossless compression? What type of compression is huffman coding? What about JPEG? Can you lossily compress a text file?
  With lossless compressions, you can re-obtains what you had originally when it's uncompressed again. You cannot lossily compress a text file.
- How big are the patterns found by Huffman Coding? What kind of files does this lend itself to?
  The longest code would be 8 bits because there are 256 characters. Since 8 bits is the maximum, the file will be smaller if the file isn't too random and equally contains all 256 characters. Mostly you will have a smaller file.
- Take a picture on your phone. What resolution is the picture? How much space does it take up in your storage (in bytes)?
  The picture is 12 MP. 3024 x 4032. It takes up 2000000 bytes.
- If each pixel takes 3 bytes, how many bytes would you expect the picture you took to take up? Why do you think that the image you took is smaller?
  It would take 36578304 bytes. It is probably smaller because there are compression algorithms employed to make it smaller.
- What is the compression ratio of the picture you just took? To get this, divide the actual size of the image by the expected size from the question above. You should not get a number above 1.
  0.0547.
- Do you expect to be able to compress the image you just took a second time with your Huffman program? Why or why not?
  No because you can't encode something a second time. If you do, you'll lose the tree.

- Are there multiple ways to achieve the same smallest file size? Explain why this might be.
  Yes you can use the lossless compression algorithm LPZ.
- When traversing the code tree, is it possible for a internal node to have a symbol?
  No because only a leaf would have a symbol. Otherwise, there would be prefix codes for multiple symbol representations.
- Why do we bother creating a histogram instead of randomly assigning a tree.
  We need to keep track of frequencies to make sure that we are getting the smallest tree.
- Relate this Huffman coding to Morse code. Why does Morse code not work as a way of writing text files as binary? What if we created more symbols for things like spaces and newlines?
  Morse code only deals with alphanumeric characters. Huffman can use anything that is byte representational.
- Using the example binary, calculate the compression ratio of a large text of your choosing

Testing

List what you will do to test your code. Make sure this is comprehensive. [1] Be sure to test inputs with delays and a wide range of files/characters.

- Enter in an invalid file name
- Enter just -i (only input)
- Enter just -o (only output)
- Enter -h (print help message)
- Enter valid input and output file names for both encoding and decoding

How to Use the Program

Audience: Write this section for the user of your program. You are answering the basic question, "How do I use this thing?". Don't copy the assignment exactly; explain this in your own words. This section will be longer for a more complicated program and shorter for a less complicated program. You should show how to compile and run your program. You should also describe any optional flags or inputs that your program uses, and what they do.

Process to encode:

There is a folder called test_files which contains multiple text files for compression. In order to compile the program the user needs to enter "make" into the terminal. To run the program, the user has to type in ./huff and then -i for input followed by the name of the input file (can be found in the test files folder). Then, type -o to write to an output file (ends with .huff), and -h to print a help message.

Process to decode:

This is a similar process as the encoding one except put in the input file that you would like to be decoded and the output file where you want the decoded message to be printed.

[1] This question is a whole lot more vague than it has been the last few assignments. Continue to answer it with the same level of detail and thought.

Program Design

Audience: Write this section for someone who will maintain your program. In industry you maintain your own programs, and so your audience could be future you! List the main data structures and the main algorithms. You are answering the basic question, "How is this thing organized so that I can have a chance of fixing it?". This section will be longer for a more complicated program and shorter for a less complicated program.
There are 6 main files.
The first is **bitreader.c** which is responsible for taking in file input and reading it bit by bit. It allocates memory for the BitReader and opens the file. It contains a function to read the bit and multiple functions for reading 8 bits at a time, 16 bits, and 32 bits.
The second file is **bitwriter.c** which writes to a file. It allocates memory for BitWriter, opens the file, creates a function to write each bit, and has separate functions to write 8 bits, 16 bits, and 32 bits at a time.
The third file is **node.c**. This file is responsible for creating nodes, freeing them, printing them, and printing the tree that consists of multiple nodes.
The fourth file is **pq.c** which uses both the node and bitwriter files. This file creates a priority queue which stores pointers from the trees. Alongside the priority queue, the file also uses the linked list data structure. This structure is used to keep track of whether an element is greater than the another to know where to place it in the tree.
The fifth file is **huff.c**. This is the file that is responsible for the encoding process. It uses all of the previous files to achieve this because it traverses through the input and creates a histogram to keep track of all the different characters. It then puts the contents into code table and writes to the tree. Finally, it compresses the file.
The sixth and final file is **dehuff.c**. This file works with decompressing the input file.

Pseudocode

Give the reader a top down description of your code! How will you break it down? What features will your code have? How will you implement each function.

bitreader.c:

```
create BitReader structure

BitReader *bit_read_open(const char *filename) {
    open file
    check for errors
    reader->underlying_stream = f;
    reader->byte = 0;
    reader->bit_position = 8;
    return reader
}

void bit_read_close(BitReader **pbuf) {
    free(*pbuf)
    *pbuf = NULL
}

uint8_t bit_read_bit(BitReader *buf) {
    check if buf->bit_position > 7
        buf->byte = (uint8_t) c;
        buf->bit_position = 0;
    get the bit numbered bit_position from byte
    buf->bit_position += 1;
    return bit;
}
```

```
uint8_t bit_read_uint8(BitReader *buf) {
    loop through 8 times
        call bit_read to read through buffer
        put bit into word
    return word;
}

uint16_t bit_read_uint16(BitReader *buf) {
    loop through 16 times
        call bit_read to read through buffer
        put bit into word
    return word;
}

uint32_t bit_read_uint32(BitReader *buf) {
    loop through 32 times
        call bit_read to read through buffer
        put bit into word
    return word;
}
```

bitwriter.c:

```
create BitWriter structure

BitWriter *bit_write_open(const char *filename) {
    Open file for writing in binary mode
    Allocate memory for bitwriter
    Close file

    writer->underlying_stream = f;
    writer->byte = 0;
    writer->bit_position = 0;

    return writer;
}

void bit_write_close(BitWriter **pbuf) {
    Use fputc to put buf->byte into buf->underlying_stream
    fclose(buf->underlying_stream);
    free(buf);
    *pbuf = NULL;
}

void bit_write_bit(BitWriter *buf, uint8_t bit) {
    Check if pit position is greater than 7
        fputc(buf->byte, buf->underlying_stream);
        buf->byte = 0;
        buf->bit_position = 0;
    Set the bit at bit_position of the byte to the value of bit
    buf->bit_position += 1;
}

void bit_write_uint8(BitWriter *buf, uint8_t byte) {
    Loop through 8 times
        uint8_t bit = (byte >> i) & 0x01;
        bit_write_bit(buf, bit);
```

```
}

void bit_write_uint16(BitWriter *buf, uint16_t x) {
    Loop through 16 times
        uint8_t bit = (byte >> i) & 0x01;
        bit_write_bit(buf, bit);
}

void bit_write_uint32(BitWriter *buf, uint32_t x) {
    Loop through 32 times
        uint8_t bit = (byte >> i) & 0x01;
        bit_write_bit(buf, bit);
}
```

node.c:

```
Node *node_create(uint8_t symbol, uint32_t weight) {
    Allocate memory for the new node
    Create new node symbol
    Create new node weight
    Create new node left neighbor
    Create new node right neighbor
    Reutrn new node
}

void node_free(Node **pnode) {
    Free left of node
    Free right of node
    Free node
    Pointer of node = NULL
}

void node_print_node(Node *tree, char ch, int indentation) {
    Print each node by calling function recursively
}

void node_print_tree(Node *tree) {
    node_print_node(tree, '<', 2);
}
```

pq.c:

```
Create ListElement structure
Create PriorityQueue structure

PriorityQueue *pq_create(void) {
    Allocate memory for the queue
    Initialize empty queue
}

void pq_free(PriorityQueue **q) {
    Free queue pointer
}

bool pq_is_empty(PriorityQueue *q) {
    return q->list == NULL;
}
```

```
bool pq_size_is_1(PriorityQueue *q) {
    return q->list != NULL && q->list->next == NULL;
}

bool pq_less_than(ListElement *e1, ListElement *e2) {
    Check if e1 weight is less than e2 weight
        return true
    Check if e1 weight is greater than e2 weight
        return false
    Check if e1 weight is equal to e2 weight
        return e1->tree->symbol < e2->tree->symbol
}

void enqueue(PriorityQueue *q, Node *tree) {
    Allocate memory for new element
    new_element->tree = tree;
    If the new element is the first element or only element
        new_element->next = q->list;
        q->list = new_element;
    If new element is inserted in middle of queue
        current element = current->next;
    new_element->next = current->next;
    current->next = new_element;
}

Node *dequeue(PriorityQueue *q) {
    Node *tree = q->list->tree;
    ListElement *temp = q->list;
    q->list = q->list->next;
    free(temp);
    return tree;
}
```

huff.c:

```
Create Code structure
uint32_t fill_histogram(FILE *fin, uint32_t *histogram){
    get bytes from the file
    add byte to histogram
    increment file size
    return file size
}

Node *create_tree(uint32_t *histogram, uint16_t *num_leaves) {
    Create priority queue
    Add symbol and weight to the node while going through histogram contents
    While there are elements in the queue
        Node *left = dequeue(pq);
        Node *right = dequeue(pq);
        Node *new_node = node_create(0, left->weight + right->weight);
        new_node->left = left;
        new_node->right = right;
        enqueue(pq, new_node);
    Dequeue pq
    Free pq
}

void fill_code_table(Code *code_table, Node *node, uint64_t code, uint8_t code_length) {
```

```
    If leaf
        fill_code_table(code_table, node->left, code, code_length + 1);
        code |= (uint64_t) 1 << code_length;
        fill_code_table(code_table, node->right, code, code_length + 1);
    If internal node
        code_table[node->symbol].code = code;
        code_table[node->symbol].code_length = code_length;
}

void huff_write_tree(BitWriter *outbuf, Node *node) {
    If leaf
        bit_write_bit(outbuf, 1);
        bit_write_uint8(outbuf, node->symbol);
    If internal node
        bit_write_bit(outbuf, 0);
        huff_write_tree(outbuf, node->left);
        huff_write_tree(outbuf, node->right);
}

void huff_compress_file(BitWriter *outbuf, FILE *fin, uint32_t filesize, uint16_t num_leaves,
     Node *code_tree, Code *code_table) {
    Node *code_tree, Code *code_table) {
    bit_write_uint8(outbuf, 'H');
    bit_write_uint8(outbuf, 'C');
    bit_write_uint32(outbuf, filesize);
    bit_write_uint16(outbuf, num_leaves);
    huff_write_tree(outbuf, code_tree);

    while bytes in file
        update code
        update code length
    close output file
}

int main(int argc, char *argv[]) {
    Get input from command line
    Check for options: i, o, h
    Open file

    uint32_t histogram[256] = { 0 };
    uint32_t filesize = fill_histogram(fin, histogram);
    uint16_t num_leaves = 0;
    Node *code_tree = create_tree(histogram, &num_leaves);
    Code *code_table = calloc(256, sizeof(Code));

    Call fill_node_table
    Call huff_compress_file

    Free code tree
    Close input file
    Free output bitwriter
    Free code table
}
```

dehuff.c:

```
def dehuff_decompress_file(fout, inbuf){
    read uint8_t type1 from inbuf
```

```
    read uint8_t type2 from inbuf
    read uint32_t filesize from inbuf
    read uint16_t num_leaves from inbuf
    assert(type1 == 'H')
    assert(type2 == 'C')
    num_nodes = 2 * num_leaves - 1
    Node *node
    for i in range(0, num_nodes):
        read one bit from inbuf
        if bit == 1:
            read uint8_t symbol from inbuf
            node = node_create(symbol, 0)
        else:
            node = node_create(0, 0)
            node->right = stack_pop()
            node->left = stack_pop()
        stack_push(node)
    Node *code_tree = stack_pop()
    for i in range(0, filesize):
        node = code_tree
        while true:
            read one bit from inbuf
            if bit == 0:
                node = node->left
            else:
                node = node->right
            if node is a leaf:
                break
        write uint8 node->symbol to fout
}

int main(int argc, char *argv[]) {
    Read the input
    dehuff_decompress_file(fout, inbuf);

    bit_read_close(&inbuf);
    fclose(fin);
    fclose(fout);

    return 0;
}
```

Function Descriptions

For each function in your program, you will need to explain your thought process. This means doing the following Do not simply use your code to describe this. This section should be readable to a person with little to no code knowledge. **DO NOT JUST PUT THE FUNCTION SIGNATURES HERE. MORE EXPLANATION IS REQUIRED.**

bitreader.c

- Function: bit_read_open. Inputs: *filename. Outputs: reader. Purpose: Reads a file input and allocates memory to a reader.
- Function: bit_read_close. Inputs: *pbuf. Outputs: none. Purpose: Frees pointer to buffer.
- Function: bit_read_bit. Inputs: *buf. Outputs: bit. Purpose: reads the bit one by one.
- Function: bit_read_uint8. Inputs: *buf. Outputs: byte. Purpose: reads through 8 bits at a time.

- Function: bit_read_uint16. Inputs: *buf. Outputs: word. Purpose: reads through 16 bits at a time.
- Function: bit_read_uint32. Inputs: *buf. Outputs: word. Purpose: reads through 32 bits at a time.

bitwriter.c

- Function: bit_write_open. Inputs: filename. Outputs: writer. Purpose: Creates a writer struct so that it can write to an output file.
- Function: bit_write_close. Inputs: **pbuf. Outputs: none. Purpose: frees the writer pointer.
- Function: buf_write_bit. Inputs: *buf, bit. Outputs: updated bit position. Purpose: writes to the output file one bit at a time.
- Function: bit_write_uint8. Inputs: *buf, x. Outputs: none. Purpose: writes through 8 bits at a time.
- Function: bit_write_uint16. Inputs: *buf, x. Outputs: none. Purpose: writes through 16 bits at a time.'
- Function: bit_write_uint32. Inputs: *buf, byte. Outputs: none. Purpose: writes through 32 bits at a time.

node.c

- Function: node_create. Inputs: symbol, weight. Outputs: new node. Purpose: Creates a new node by allocating memory for it and creating the weight, symbol, left neighbor, and right neighbor.
- Function: node_free. Inputs: **pnode. Outputs: none. Purpose: frees the node.
- Function: node_print_node. Inputs: *tree, ch, indentation. Outputs: printed tree. Purpose: prints each node
- Function: node_print_tree. Inputs: *tree. Outputs: printed tree. Purpose: calls the node_print_node function to print each node for the entire tree.

pq.c

- Function: pq_create. Inputs: none. Outputs: priority queue. Purpose: Creates the priority queue.
- Function: pq_free. Inputs: priority queue. Outputs: none. Purpose: frees the prirority queue.
- Function: pq_is_empty. Inputs: *q. Outputs: empty queue. Purpose: returns an empty queue.
- Function: pq_size_is_1. Inputs: *q. Outputs: size. Purpose: checks if the size of the queue is 1.
- Function: pq_less_than. Inputs: element1, element2. Purpose: checks if element 1 is less than element 2.
- Function: enqueue. Inputs: *q, *tree. Outputs: updated queue. Purpose: queues an item into the priority queue and updates the rest of it accordingly.
- Function: dequeue. Inputs: *q. Outputs: tree. Purpose: removes an element from the queue.
- Function: pq_print. Inputs: *q. Outputs: printed queue. Purpose: prints the queue.

huff.c

- Function: fill_histogram. Inputs: *fin, *histogram. Outputs: filesize. Purpose: takes bytes and fills them into the histogram.
- Function: create_tree. Inputs: *histogram, *num_leaves. Outputs: root. Purpose: takes the nodes and puts them into the tree. Also updates the size and the weights.

- Function: fill_code_table. Inputs: Code *code_table, Node *node, uint64_t code, uint8_t code_length. Outputs: code table. Purpose: recursively fills the code table with the corresponding nodes.
- Function: huff_write_tree. Inputs: *outbuf, *node. Outputs: tree. Purpose: Updates tree with symbols, left node, right node, and weight.
- Function: huff_compress_file. Inputs: *outbuf, *fin, filesize, num leaves. Outputs: compressed file. Purpose: Writes to the output by calling the bitwriter functions and then closes the file.
- Function: main. Inputs: argc, argv. Outputs: compressed file. Purpose: reads the file input and calls the huff functions.

dehuff.c

- Function: dehuff_decompress_file. Inputs: *fout, *inbuf. Outputs: decompressed content. Purpose: reads the file input and writes the corresponding elements onto the output file.
- Function: main. Inputs: argc, argv. Outputs: decompressed file contents. Purpose: reads in inputs and error checks. Calls function to decompress. Returns 0 if successful.

References